Object Oriented Programming I

Table of contents

Outline 1
Why OOP? 1
The four pillars of object orientation 8
The mechanics of each pillar 9

See csse2002[1] for staff, assessment, and course logistics.

Outline

1. Objects, Classes, & Object-Oriented Programming
2. Encapsulation
3. Inheritance
4. Abstraction
5. Polymorphism (Week 3 — java-polymorphism[2])

Why OOP?

The case for objects rests on **one running example grown in four stages**: a bank, written first with nothing but static fields and static methods, then rewritten with objects. Watching what breaks at each stage is the argument for OOP. The bank carries only that argument. Inheritance has its own example, `Employee`, and abstraction another, `PaymentProcessor`/`Payment`. Polymorphism is left whole to Week 3.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-polymorphism.pdf

Stage 1 — one global balance

```
public class BankBasic {
    static double bankBalance = 0; // Single balance for the entire bank

    public static void deposit(double amount) {
        bankBalance += amount;
    }

    public static void withdraw(double amount) {
        if (bankBalance >= amount) {
            bankBalance -= amount;
        } else {
            System.out.println("Insufficient funds.");
        }
    }

    public static void checkBalance() {
        System.out.println("Bank Balance: $" + bankBalance);
    }

    public static void main(String[] args) {
        deposit(5000);
        withdraw(2000);
        checkBalance(); // Output: Bank Balance: $3000.0
    }
}
```

The listing has two kinds of thing in it: the field `bankBalance` is a **variable**, x, and the three methods are **functions** of it, $f(x)$. That framing — *free variables on one side, free functions on the other* — is the shape the next three stages keep running into.

There is one balance for the entire bank. There is no way to represent a second account, let alone a second customer.

Stage 2 — parallel arrays of accounts

```
public class BankWithAccounts {
    static final int MAX_ACCOUNTS = 10;
    static String[] accountNumbers = new String[MAX_ACCOUNTS];
    static double[] balances = new double[MAX_ACCOUNTS];
    static int accountCount = 0;

    public static void createAccount(String accountNumber, double initialBalance) {
     ↪  ... }
```

```
    public static int findAccountIndex(String accountNumber) { ... }
    public static void deposit(String accountNumber, double amount) { ... }
    public static void withdraw(String accountNumber, double amount) { ... }
    public static void checkBalance(String accountNumber) { ... }

    public static void main(String[] args) {
        createAccount("A123", 1000);
        createAccount("B456", 2000);

        deposit("A123", 500);
        withdraw("B456", 300);

        checkBalance("A123"); // Expected: Balance for A123: $1500.0
        checkBalance("B456"); // Expected: Balance for B456: $1700.0
    }
}
```

Every piece of account data needs its own array, and the arrays are held together only by **agreeing on the index**. `accountCount` has to be maintained by hand, and a `findAccountIndex` helper now stands between an account number and the position it names.

Stage 3 — parallel arrays of account *holders* too

```
public class BankWithHolders {
    static final int MAX_ACCOUNTS = 10;

    // Account Holder Data
    static String[] holderIds   = new String[MAX_ACCOUNTS];
    static String[] holderNames = new String[MAX_ACCOUNTS];
    static int holderCount = 0;

    // Account Data
    static String[] accountNumbers  = new String[MAX_ACCOUNTS];
    static double[] balances        = new double[MAX_ACCOUNTS];
    static String[] linkedHolderIds = new String[MAX_ACCOUNTS]; // Link accounts to
     ↪ holders
    static int accountCount = 0;

    public static void createAccountHolder(String id, String name) { ... }
    public static int findHolderIndex(String holderId) { ... }
    public static void createAccount(String accountNumber, double initialBalance,
                                     String holderId) { ... }
    public static int findAccountIndex(String accountNumber) { ... }
    public static void deposit(String accountNumber, double amount) { ... }
    public static void withdraw(String accountNumber, double amount) { ... }
```

```
    public static void checkBalance(String accountNumber) { ... }

    public static void main(String[] args) {
        createAccountHolder("001", "Jack Johnson");
        createAccountHolder("002", "Jill Smith");

        createAccount("A123", 1000, "001");
        createAccount("B456", 2000, "002");

        deposit("A123", 500);
        withdraw("B456", 300);

        checkBalance("A123"); // Output: Balance for Jack Johnson: $1500.0
        checkBalance("B456"); // Output: Balance for Jill Smith: $1700.0
    }
}
```

Adding a *second kind of thing* (holders) doubles the bookkeeping: two more parallel arrays, a second count, a second index-finder, and a third array (`linkedHolderIds`) whose only job is to remember which holder each account belongs to. Nothing in the language enforces that any of these arrays stay in step.

What that program actually looks like

The variables `a`, `c` and `x` belong to nobody. Any function may read or write any of them, and the call graph is free to cross in any direction. Nothing in the picture says which function is allowed to touch which variable — that knowledge lives only in the programmer's head.

The fix: group each variable with the functions that act on it

This is the whole idea: instead of scattering related data across parallel arrays and free functions, **bundle a variable and the functions that act on it into a single unit** — a **class**. The second figure uses a fresh cast of functions, so don't try to match the two figures member for member. Bundling the data and the functions that operate on the data into a single unit restricts direct access to some of an object's components and can prevent the accidental modification of data.

Stage 4 — the same bank, with objects

The bank is then rewritten as four classes. Arrays do not disappear — each *set* of parallel arrays collapses into a single array of objects (`BankAccount[]`, `AccountHolder[]`) held privately inside

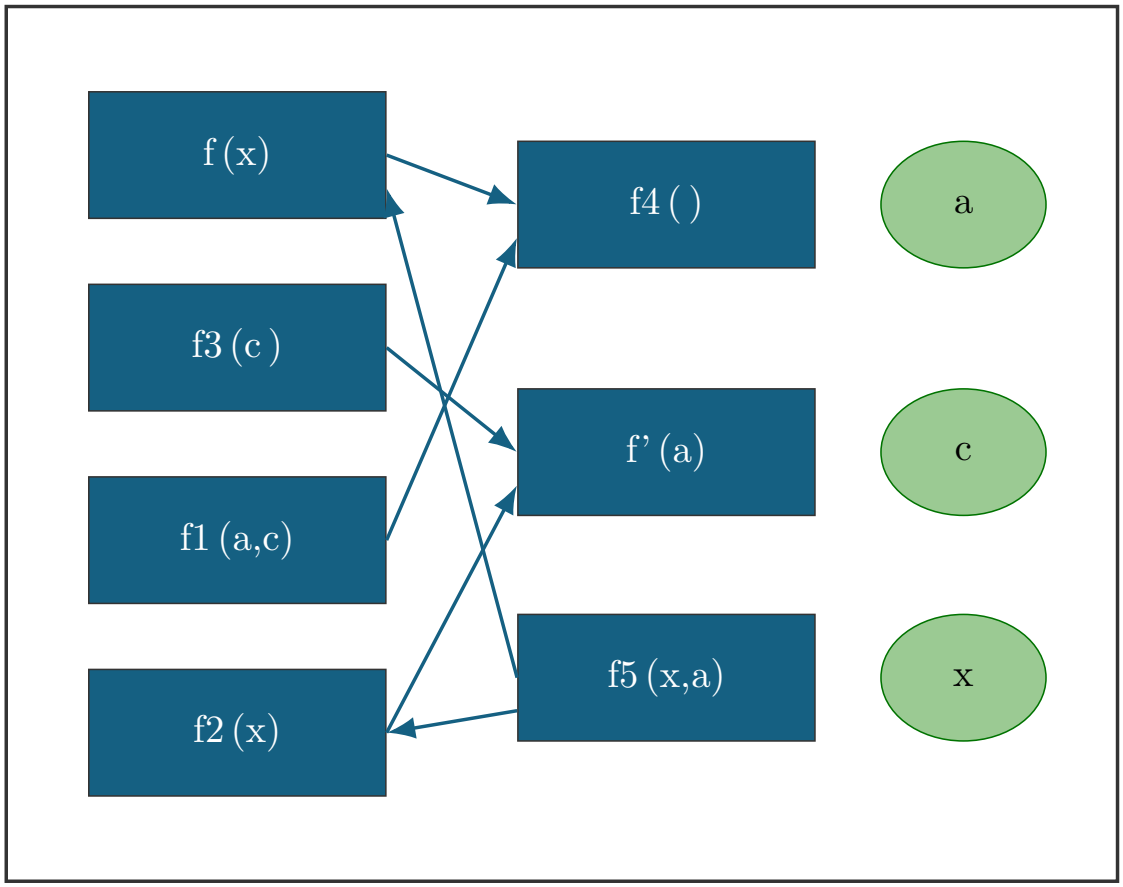

Figure 1: The procedural picture after the three bank stages: every function and every variable lives in one flat frame, and any function may call any other. The boxes are functions, the ellipses are variables and the six arrows are calls.

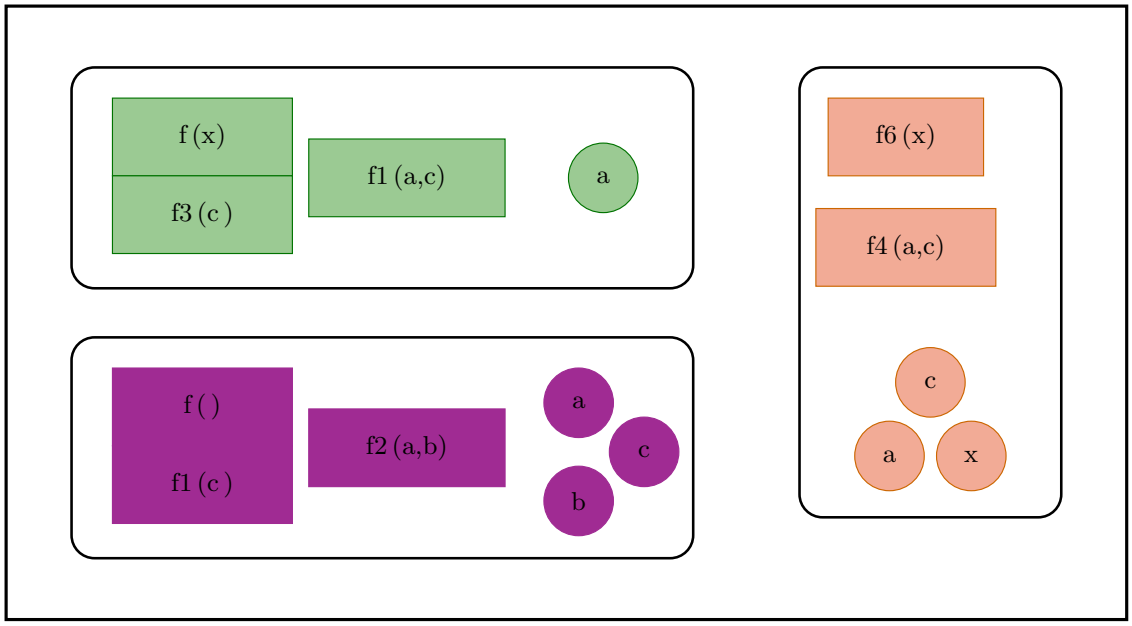

Figure 2: Three self-contained units, each owning its own data and the functions that operate on it. Each unit has its own colour. This is a fresh illustration, not a literal regrouping of the procedural figure: f6 is new, f' and f5 are gone, and f2 and f4 have different parameter lists.

`Bank`, and the per-thing data that used to be spread across those arrays becomes fields on the object:

```
public class AccountHolder {
    private String id;
    private String name;
    private String contactInfo;

    public AccountHolder(String id, String name, String contactInfo) {
        this.id = id;
        this.name = name;
        this.contactInfo = contactInfo;
    }
    public String getId() { return id; }
    public String getName() { return name; }
    public String getContactInfo() { return contactInfo; }
}

public class BankAccount {
    private String accountNumber;
    private double balance;
    private AccountHolder accountHolder; // Association with AccountHolder

    public BankAccount(String accountNumber, double initialBalance,
                       AccountHolder accountHolder) { ... }

    public void deposit(double amount) { balance += amount; }
    public void withdraw(double amount) { ... }
    public double checkBalance() { return balance; }
    public String getAccountNumber() { return accountNumber; }
    public AccountHolder getAccountHolder() { return accountHolder; }
}

public class Bank {
    private static final int MAX_ACCOUNTS = 10;
    private BankAccount[] accounts = new BankAccount[MAX_ACCOUNTS];
    private int accountCount = 0;
    private AccountHolder[] accountHolders = new AccountHolder[MAX_ACCOUNTS];
    private int holderCount = 0;

    public void createAccountHolder(String id, String name, String contactInfo) { ...
     ↪  }
    private AccountHolder findAccountHolder(String id) { ... }
    public void createAccount(String accountNumber, double initialBalance,
                              String holderId) { ... }
    private BankAccount findAccount(String accountNumber) { ... }
    public void deposit(String accountNumber, double amount) { ... }
    public void withdraw(String accountNumber, double amount) { ... }
    public void checkBalance(String accountNumber) { ... }
```

```
    public void displayAllAccounts() { ... }
}
```

and the driver becomes a script against one `bank` object:

```
public class OOPBanking {
    public static void main(String[] args) {
        Bank bank = new Bank();

        // Creating account holders
        bank.createAccountHolder("001", "Jack Johnson", "jack@uq.com.au");
        bank.createAccountHolder("002", "Jill Smith", "jill@uq.com.au");

        // Creating accounts linked to account holders
        bank.createAccount("A123", 1000, "001");
        bank.createAccount("B456", 2000, "002");

        // Performing transactions
        bank.deposit("A123", 500);
        bank.withdraw("B456", 300);

        // Checking balances
        bank.checkBalance("A123"); // Output: Balance for Jack Johnson: $1500.0
        bank.checkBalance("B456"); // Output: Balance for Jill Smith: $1700.0

        // Displaying all accounts
        bank.displayAllAccounts();
    }
}
```

Three things changed, and they are the payoff:

- Every field is now `private` — the arrays that used to be `static` and open to the world are hidden inside `Bank`.
- `findAccountHolder` and `findAccount` are `private`: index bookkeeping became an internal detail rather than part of the public surface.
- The "which holder owns this account" link is a **reference field** on `BankAccount` (`private AccountHolder accountHolder`), not a third parallel array. This kind of link is an *association*.

The four pillars of object orientation

1. **Encapsulation** — see java-encapsulation[3]

[3] courses/csse2002/java-encapsulation.pdf

2. **Inheritance** — see java-inheritance[4]
3. **Abstraction** — see java-abstraction[5]
4. **Polymorphism** — Week 3 (java-polymorphism[6], 2026-03-12-object-oriented-programming-ii[7])

The mechanics of each pillar

The mechanics under each pillar are written up in the concept notes:

Topic	Note
classes vs objects, constructors and the default constructor	java-encapsulation[8]
access modifiers (`private`, `public`, `protected`, default)	java-encapsulation[9]
getters and setters	java-encapsulation[10]
the `static` keyword — class variables and static methods	java-encapsulation[11]
class invariants	java-encapsulation[12]
method signatures and why they must be unique	java-encapsulation[13]
`extends`, `super`, method overriding	java-inheritance[14]
interfaces and abstract classes	java-abstraction[15]

Two items are **homework**:

- `protected` access — read DigitalOcean's Java access modifiers tutorial.
- **Interface vs abstract class** — two Stack Overflow threads to read; picked up in 2026-03-12-object-oriented-programming-ii[16].

Preconditions and postconditions come in Week 3 — see java-specification[17].

[4] courses/csse2002/java-inheritance.pdf
[5] courses/csse2002/java-abstraction.pdf
[6] courses/csse2002/java-polymorphism.pdf
[7] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[8] courses/csse2002/java-encapsulation.pdf
[9] courses/csse2002/java-encapsulation.pdf
[10] courses/csse2002/java-encapsulation.pdf
[11] courses/csse2002/java-encapsulation.pdf
[12] courses/csse2002/java-encapsulation.pdf
[13] courses/csse2002/java-encapsulation.pdf
[14] courses/csse2002/java-inheritance.pdf
[15] courses/csse2002/java-abstraction.pdf
[16] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[17] courses/csse2002/java-specification.pdf